ECO-LLM: Orchestration for Domain-specific Edge-Cloud Language Models

Prasoon Patidar[1] **Alex Crown**[2] **Kevin Hsieh**[2] **Yifei Xu**[2]
Tusher Chakraborty[2] **Ranveer Chandra**[2] **Yuvraj Agarwal**[1]
[1]*Carnegie Mellon University* [2]*Microsoft Research*

Abstract

The remarkable performance of Large Language Models (LLMs) has inspired many applications, which often necessitate edge-cloud collaboration due to connectivity, latency, and cost considerations. Traditional methods primarily focus on selecting the best LLM model for optimizing performance, while neglecting the critical interplay between the components of the LLM serving pipeline (context retrieval, query preprocessing, etc.) or the changing latency and cost constraints. We introduce ECO-LLM (Edge-Cloud Orchestrator for LLMs), a novel system that reframes this problem as a joint optimization challenge and solves it by systematically exploring component configurations and dynamically selecting optimal strategies at the query level.

ECO-LLM consists of two components: (1) the *ECO-LLM Emulator*, which efficiently explores the vast configuration space utilizing query clustering and pareto-optimal path selection, gathering domain-specific performance metrics without exhaustive evaluation; and (2) the *ECO-LLM Runtime*, which leverages these metrics to dynamically select optimal resolution strategies for user queries while meeting user-defined Service Level Objectives (SLOs).

We evaluate ECO-LLM across five diverse domains on four hardware platforms. Compared to state-of-the-art model routing approaches, ECO-LLM reduces costs by 60% and latency by up to 6x while maintaining consistent performance (73-87% accuracy) across domains, whereas model-routing approaches exhibit high variance (54-85%)—degrading significantly on domains that require coordinated query preprocessing rather than model selection alone. ECO-LLM consistently meets user-defined SLOs while maintaining stable accuracy, and its stratified exploration achieves near-equivalent performance with up to 65% fewer evaluations—making systematic domain adaptation practical.

1 Introduction

The rapid advances in Large Language Models (LLMs) has led to the proliferation of AI assistants [6, 16, 23, 29]. These AI assistants for smart homes, connected cars, and mobile devices benefit significantly from edge deployments. For instance, smart home assistants process voice commands locally for privacy, driving assistants require real-time data processing to navigate safely, minimizing reliance on cloud connectivity. Additionally, edge processing can reduce the cost of LLM inference on the cloud, a major concern for applications requiring frequent interactions with LLMs.

Despite the potential, developing an edge-AI assistant is challenging due to an ever expanding design space. For example, an end-to-end *serving pipeline* can choose from a selection of LLM [11–13], retrieval-augmented generation (RAG) [26] for fetching relevant context, and query/context refinement [42, 51]. Each of these components represents an active research area with rapid advances [15, 49, 52]. As we show, the selection of these components critically impacts overall performance and must consider factors such as edge device capabilities, domain-specific requirements, and user-defined constraints for cost and latency.

Current approaches to address this challenge primarily focus on optimizing individual components in isolation. Solutions such as RouteLLM [33], AutoMix [1] and HybridLLM [10] have introduced model routing systems that optimize for selecting the best model for a given query, offering simple trade-offs such as quality versus cost. However, edge AI assistants inherently require domain-specific optimization, *i.e.*, automotive assistants require rapid responses with efficient content retrieval while smart home assistants must handle ambiguous queries requiring sophisticated reasoning. A critical insight missing from existing work is that optimal configurations require simultaneous selection across *multiple pipeline* components, not just model selection. Some queries achieve higher accuracy with less context given to a powerful model, while others benefit from extensive retrieval with a smaller model. The challenge lies not just in finding these synergistic configurations, but in making domain-specific deployment efficient and systematic rather than requiring manual exploration for each new domain.

To address this gap, we present ECO-LLM, a deployment

framework that transforms edge-cloud LLM optimizations from a manual, domain-by-domain effort into a systematic process. Rather than providing a pre-trained solution, ECO-LLM automatically explores query resolution paths for each domain and hardware setup, enabling per-query selection of optimal component configurations at runtime. Our key insight is that by treating component selection as a unified optimization problem and automating the exploration process, we can efficiently discover superior pareto frontiers balancing accuracy, cost, and latency for each specific domain.

ECO-LLM consists of two complementary components: a *ECO-LLM Emulator* and a *ECO-LLM Runtime*. The *ECO-LLM Emulator* provides a systematic exploration framework that efficiently characterizes different query resolution paths for a target domain and hardware setup across multiple metrics. By intelligently sampling the vast configuration space without requiring exhaustive evaluation, it uncovers patterns that would remain hidden when optimizing components independently. Building on these insights, the *ECO-LLM Runtime* dynamically selects optimal query resolution paths in real-time, combining domain-specific query encoding with constraint-aware path selection to consistently meet user-defined SLOs. This two-stage approach makes domain adaptation practical—the emulator handles the complex exploration once per domain/hardware combination, while the runtime efficiently serves queries using the discovered strategies.

In developing ECO-LLM, we addresses several key technical challenges. The system establishes a structured framework for representing and exploring the exponential configuration space of LLM serving pipelines, enabling comprehensive performance characterization without evaluating all possible component combinations. It introduces a novel approach for critical component identification that reduces the effective search space through impact-based analysis, making real-time decisions practical. Finally, it implements a robust path selection mechanism that generalizes performance patterns to unseen queries while strictly maintaining SLO guarantees. Our framework-based approach allows organizations to deploy edge AI assistants for new domains or hardware platforms efficiently, providing fine-grained control over latency and cost without the burden of manual optimization.

We evaluate ECO-LLM across four hardware platforms, Jetson Orin Nano, M1 Pro, M4, and RTX A4500 spanning resource-constrained to high-performance edge computing, and five diverse domains: automotive diagnostics, smart home automation, agriculture, technical support, and IoT security. Our results show that M4 class platforms (30-40 TOPS, 20-32 GB RAM) are a practical deployment target for responsive edge AI assistants—achieving sub-second to few-second latencies while the Jetson Orin Nano exhibits prohibitive response times (12+ seconds) and the RTX A4500 yields diminishing returns despite higher power/cooling needs. Across domains, ECO-LLM reduces costs by 60% and latency by up to 6x compared to state-of-the-art model routing approaches while maintaining comparable accuracy. Critically, ECO-LLM delivers predictable performance (73-87% accuracy) across all domains, whereas model-routing approaches show high variance (54-85%)—degrading significantly on domains like smart home and technical support where queries benefit from coordinated preprocessing rather than model selection alone. Our ablation studies confirm that both ECO-LLM's query understanding and adaptive path selection are necessary: removing the former degrades accuracy, while removing the latter forces suboptimal cost-latency tradeoffs. Finally, ECO-LLM configurations achieve near-zero SLO violations when constraints are feasible, maintaining stable accuracy.

In summary, we make the following contributions:

1. We formulate edge-cloud LLM deployment as a joint component selection problem, demonstrating that coordinated optimization across the serving pipeline delivers consistent performance where model-routing-only approaches exhibit high variance across domains.
2. We built the ECO-LLM Emulator, an exploration framework that efficiently characterizes query resolution paths through stratified budget allocation and critical component analysis, enabling systematic domain adaptation without exhaustive evaluation.
3. We develop the ECO-LLM Runtime, which combines domain-specific query encoding with constraint-aware path selection to dynamically select optimal resolution strategies while meeting user-defined SLOs.
4. We evaluate ECO-LLM across five domains and four hardware platforms, demonstrating 60% cost reduction and up to 6× latency improvement over model-routing approaches, with predictable 73-87% accuracy and consistent SLO attainment.

2 Related Work

The growing adoption of LLMs in edge applications has led to significant research addressing various aspects of LLM serving systems. We categorize existing approaches based on their primary focus and optimization strategies, and discuss how ECO-LLM relates to these approaches.

Model Routing Systems: A significant body of work has focused on routing queries to appropriate language models based on query characteristics. Systems like RouteLLM [33] and AutoMix [1] use routers trained on human preference datasets to direct queries to models with different capabilities, primarily optimizing cost-quality tradeoffs. Other approaches such as MetaLLM [31], LLMProxy [28], and Eagle [50] have introduced different strategies for model selection and scheduling to optimize inference performance.More recent approached like OptiRoute [34] route tasks to the optimal LLM based on detailed user-defined requirements. These systems effectively address model selection as a standalone optimization problem. However, they don't explore how model selection interacts with other aspects such as context retrieval or query processing. Additionally, most of these approaches focus on cloud-based deployment, with less attention to edge

Table 1: Comparing ECO-LLM with different approaches for LLM inference. Unlike existing solutions that optimize individual components separately, ECO-LLM addresses all of the requirements of an edge-cloud orchestration framework.

Approach	Model Routing	Cost/ Latency	Context/ Query Opt.	Edge Deploy	Policy Constr.
Model Routing Solutions					
RouteLLM [33]	✓	✓/×	×/×	×	×
MetaLLM [31]†	✓	✓/×	×/×	×	×
LLMProxy [28]†	✓	✓/×	×/×	×	×
AutoMix [1]	✓	✓/×	×/×	×	×
Eagle [50]	✓	✓/✓	×/×	×	×
TensorOpera [38]	✓	✓/✓	×/×	×	×
Cascade Router [9]†	✓	✓/✓	×/×	×	×
ScaleLLM [45]	✓	×/✓	×/×	×	×
RouterDC [8]	✓	×/×	×/×	×	×
EmbedLLM [53]	✓	×/×	×/×	×	×
OptiRoute [34]	✓	✓/✓	×/×	×	✓
Routoo [30]†	✓	✓/×	×/×	×	×
Context Management Solutions					
Teola [39]†	×	×/✓	✓/✓	×	×
AutoRAG-HP [14]	×	✓/×	✓/✓	×	×
HyPA-RAG [22]	×	✓/×	✓/✓	×	×
Mentor-KD [24]	✓	✓/✓	✓/✓	×	×
Edge-Focused Solutions					
HybridLLM [10]	✓	✓/×	×/×	✓	×
EdgeShard [48]†	✓	✓/✓	×/×	✓	×
PerLLM [44]†	✓	✓/✓	×/×	✓	✓
ECO-LLM (Our Approach)	✓	✓/✓	✓/✓	✓	✓

environments.

Context Management and Query Optimization: Another research direction has explored optimizing context management and query processing in LLM systems. Frameworks like Teola [39] introduce orchestration systems that optimize dataflow representations of query workflows, while approaches like AutoRAG-HP [14] and HyPA-RAG [22] focus on hyper-parameter tuning for retrieval-augmented generation. Mentor-KD [24] takes a different approach by enhancing smaller models through knowledge distillation. These solutions provide valuable insights into optimizing specific components of the pipeline but generally consider these optimizations separately from model selection and edge deployment considerations.

Edge-Focused Deployments: Recent works have begun addressing the challenges of edge deployment for LLMs. HybridLLM [10] proposes a cost-efficient routing approach for edge environments. EdgeShard [48]† introduces a partitioning strategy that distributes model components between edge and cloud, while PerLLM [44]† utilizes a multi-arm bandit optimization to schedule inference tasks across edge and cloud. More recent advances in edge-cloud collaborative frameworks include Edge-LLM [46], EdgeFM [43], and CE-CoLLM [20]† , which explore sophisticated approaches ranging from efficient model adaptation through layerwise compression, using a foundation model for open-set recognition, and enabling collaborative edge-cloud inference strategies. While these approaches make important contributions to edge-specific deployment challenges, they focus primarily on model distribution, and collaborative inference strategies. They generally give less attention to the holistic optimization of context retrieval and query manipulation components alongside model selection, which can be critical for maintaining response quality in resource-constrained environments. These model-centric optimizations can be effectively integrated with ECO-LLM to provide domain-specific pipeline optimization, complementing their focus on efficient model execution with systematic component selection.

Research Gap and Our Contributions:

As Table 1 illustrates, existing approaches reflect a gap in the research landscape—the lack of integrative frameworks that address the interplay between these components. While each component optimization is valuable individually, their interactions remain underexplored. For instance, how context retrieval strategies should adapt to different model choices, or how query processing needs change based on edge deployment conditions. ECO-LLM's contribution lies in connecting these disparate optimization domains into a unified framework. By integrating policy-driven constraints with component-level optimizations, our work builds upon these works. This approach allows specialized systems to be composed in ways that respect their interdependencies, potentially offering a more complete perspective on LLM serving optimization for domain-specific deployments.

3 System Design

We envision a framework capable of addressing the diverse challenges and requirements of Edge AI assistants across various domains. This framework will integrate local processing (edge databases, on-device inference, caching) with cloud resources (larger language models, extensive databases). The "Edge" refers to the computing platforms one step removed from the most constrained environments (microcontrollers, basic sensors, etc.), i.e., devices that can run Small Language Models (SLMs) and host local databases, including consumer laptops with NPUs, automotive computing units, smart home hubs, and industrial edge servers—while maintaining the benefits of local processing.

Each domain presents unique hurdles: automotive assistants demand rapid, precise responses regarding vehicle systems, emphasizing efficient content retrieval, whereas smart home assistants grapple with ambiguous queries necessitating more sophisticated reasoning. Furthermore, requirements can vary even within a single domain, depending on the query. For instance, an automotive assistant in a smart car benefits from web data retrieval for weather updates and navigation, but relies on local, specialized databases for vehicle-specific queries.

[1]† arXiv technical reports

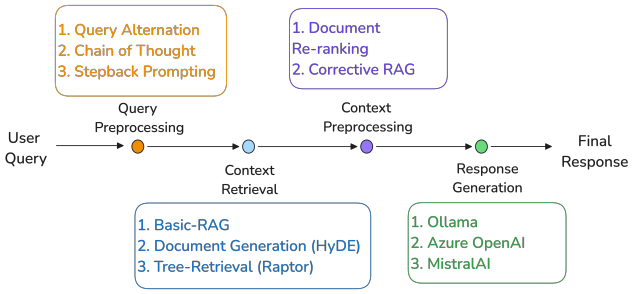


Figure 1: Sequence of components (or *query resolution path*) to resolve a user query with edge AI assistants.

3.1 Problem Formulation

We define a query resolution path P as the sequence of operations that transforms a user query into a final response (See Figure 1). Each path is an ordered tuple $P = (M_q, M_r, M_c, M_m)$, where M_q represents query processing, M_r represents retrieval strategy, M_c represents context processing, and M_m represents model selection. Each module M_i has multiple potential implementations I_i and parameter configurations $C_{i,j}$ for each implementation $j \in I_i$. A concrete path instantiation involves selecting both implementations and configurations: $P = ((q,\theta_q),(r,\theta_r),(c,\theta_c),(m,\theta_m))$.

The space of all possible paths $\mathcal{P}$ grows combinatorially with the number of modules, implementations, and configurations (see Table 2):

$$|\mathcal{P}| = \prod_{i \in \{q,r,c,m\}} \sum_{j \in I_i} |C_{i,j}| \tag{1}$$

For each query $q \in Q$ and path $P \in \mathcal{P}$, we define three key performance metrics:

$$\begin{aligned} \text{Accuracy}(q,P) &: Q \times \mathcal{P} \to [0,1] \\ \text{Latency}(q,P) &: Q \times \mathcal{P} \to \mathbb{R}^+ \\ \text{Cost}(q,P) &: Q \times \mathcal{P} \to \mathbb{R}^+ \end{aligned} \tag{2}$$

We measure latency as Time to First Token (TTFT), which denotes the time until the user receives the first response token. TTFT includes query preprocessing, context retrieval and postprocessing, model inference, and ECO-LLM's path selection overhead. We estimate cost based on token usage:

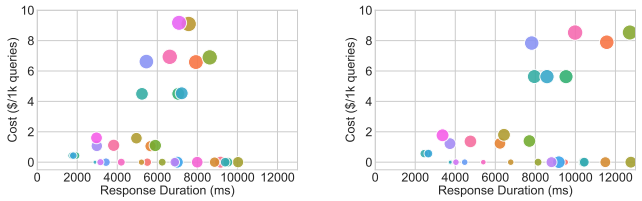
$$\mathcal{C}(q,P) = \alpha_P \cdot |q| + \beta_P \cdot \text{max\_tokens} \tag{3}$$

where α_P and β_P are model-specific pricing for input and output tokens.

Service Level Objectives (SLOs) impose constraints on latency and cost:

$$\begin{aligned} \text{Latency}(q,P) &\le \mathcal{L}_{\max} \\ \text{Cost}(q,P) &\le \mathcal{C}_{\max} \end{aligned} \tag{4}$$

We identify two challenges in edge-cloud orchestration:



(a) Smart Home Assistant (b) Automotive Assistant

Figure 2: Performance variations in query resolution paths for smart home and Automotive AI assistants due to differences in domain-specific queries. Each color represents a single path (or configuration), and the marker size represents the quality of response. Different paths have optimal response quality for different domains.

1) Given a domain-specific dataset D of representative queries, we must efficiently explore the performance characteristics of different paths within a given budget B.

$$\begin{aligned} \max_{P_{subset} \subset \mathcal{P}} \quad & \text{InformationGain}(P_{subset}) \\ \text{s.t.} \quad & |P_{subset}| \le B \end{aligned} \tag{5}$$

2) For a new, previously unseen query q_{new}, we need to select a path that satisfies SLO constraints while maximizing accuracy in real time:

$$\begin{aligned} \max_{P \in \mathcal{P}} \quad & \text{Accuracy}(q_{new},P) \\ \text{s.t.} \quad & \text{Latency}(q_{new},P) \le \mathcal{L}_{\max} \\ & \text{Cost}(q_{new},P) \le \mathcal{C}_{\max} \end{aligned} \tag{6}$$

A key insight is that the effectiveness of resolution paths varies significantly across domains (see Figure 2). The same path yields different latency-cost-accuracy tradeoffs for smart homes versus automotive assistants. Moreover, even within a single domain, optimal paths vary significantly by query characteristics. While simple heuristics—such as routing complex queries to cloud models or always using maximum context retrieval—can achieve reasonable accuracy, they could incur much higher costs and are unsuitable for scenarios with strict latency requirements. Finally, the optimization challenge is compounded by component interdependencies where model performance depends critically on query preprocessing and retrieved context.

3.2 *ECO-LLM Emulator*

The *ECO-LLM Emulator* addresses the challenge of efficiently exploring the exponential space of query resolution paths to identify optimal configurations for domain-specific queries. Instead of implementing the components themselves, *ECO-LLM Emulator* provides a framework for systematically evaluating different combinations of configurations.

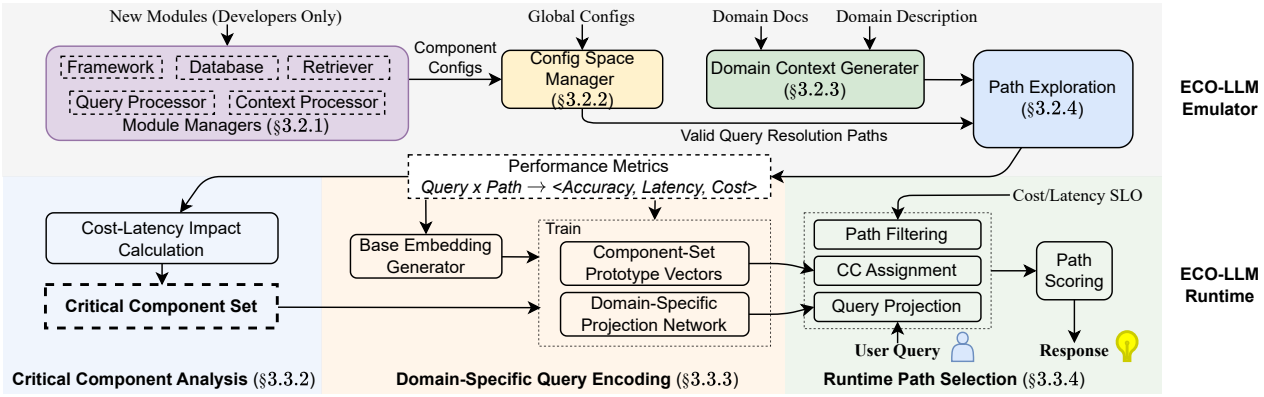


Figure 3: Overall Architecture of ECO-LLM, showing the components of the *ECO-LLM Emulator* and the *ECO-LLM Runtime*.

Table 2: Examples of implementations ($j \in I_i$) for each module M_i and their configurable parameters. Parameters can be defined as static values, parameter ranges to sweep through during exploration, or dynamically resolved at runtime via methods (e.g., available models). Each query resolution path $P = ((q, \theta_q), (r, \theta_r), (c, \theta_c), (m, \theta_m))$ represents a specific implementation and parameter configuration choice. The combinatorial explosion of options leads to an exponential growth in the configuration space, making systematic exploration critical.

Module Type	**Implementations**	**Configurable Options**
Model Frameworks	Ollama [32]	model name, temperature, context length
	Azure OpenAI [17]	api key, deployment name, temperature, max tokens, presence penalty
Query Processing (Pre-retrieval)	Chain of Thought [41]	reasoning depth, intermediate steps, verification strategy
	Step-back Prompting [51]	abstraction level, context breadth
Context Retrieval	Basic RAG [25]	chunk size, overlap ratio, top-k
	HyDE [21]	hypothesis model, hypothesis count, aggregation strategy
Context Processing (Post-retrieval)	Corrective RAG [42]	correction threshold, iteration limit, confidence score
	Document Reranking [5]	reranking model, score threshold, cross-attention weight

3.2.1 Modular Architecture

Given that new language models, retrieval techniques, and processing methods emerge monthly, we designed *ECO-LLM Emulator* to be extensible. The architecture consists of five standardized interfaces implemented as module managers: Framework Manager (M_m) for language model interactions, Database Manager for vector storage, Query Processor Manager (M_q) for query transformations, Retriever Manager (M_r) for context extraction, and Context Processor Manager (M_c) for post-retrieval refinement. This design offers three key advantages: First, it enables easy integration of emerging technologies, i.e., new retrieval methods or language models can be added by implementing the appropriate interface without modifying other system components. Second, it facilitates systematic experimentation by allowing different combinations of implementations to be evaluated under consistent conditions. Third, it provides abstraction across deployment environments, with each manager supporting both edge and cloud implementations through the same interface. The modules interact through well-defined boundaries while leveraging each other's capabilities—for example, retrievers can use language models via the Framework Manager to enhance search effectiveness. This design ensures that *ECO-LLM Emulator* can continuously incorporate state-of-the-art components while maintaining systematic evaluation capabilities.

3.2.2 Configuration Space Management

ECO-LLM Emulator employs a hierarchical configuration system with component-level and global-level specifications to keep configuration management tractable. At the component level, each implementation for a module specifies three types of parameters: static values for standard operation, parameter ranges to sweep through during experimentation, and dynamic values resolved at runtime. For example, a language model framework might include static temperature values, model context lengths to sweep through, and dynamically resolved model names based on what is available at runtime. At the global level, configurations define which implementations to include for each module, execution settings, and evaluation strategies. The global configuration can also specify null options to allow the system to skip certain operations entirely.

3.2.3 Domain Context Generation

Many domains such as automotive diagnostics or interactions with industrial equipment and medical devices have technical documentation but lack query datasets for evaluating LLM pipelines. Prior approaches propose creating these query datasets manually by domain experts which is time consuming and tedious. *ECO-LLM Emulator* addresses this query dataset creation through its *Context Generator*, which creates training

datasets from domain documentation (e.g., vehicle manuals) and a brief domain description.

The Context Generator uses LLMs to produce questions from documentation through content extraction, chunking, and prompt-guided query generation. It creates six query types targeting different information needs: retrieval questions for specific facts, explanation questions for causal relationships, analysis questions for multi-factor reasoning, solving questions for troubleshooting procedures, comparison questions for evaluating alternatives, and recommendation questions for optimizing under constraints.

For example, retrieval questions target specific information (*"What is the implication of unauthorized modifications on operating authority?"*), while analysis questions require reasoning across multiple factors (*"What are safety implications if the Reverse Brake Assist warning persists despite troubleshooting?"*). Recommendation questions consider multiple constraints (*"How should I schedule charging to minimize costs while ensuring morning readiness?"*). These queries combine multiple system aspects rather than following simple templates. Each query Q_i includes metadata (T_i, C_i, E_i) representing query type, expected context/reference answer, and evaluation guidelines for automated response quality assessment. Application owners may optionally review generated answers to further improve quality. This approach requires only documentation that organizations deploying domain-specific AI assistants typically already possess.

3.2.4 Path Exploration

ECO-LLM Emulator characterizes query resolution paths across the configuration space defined in Equation 1. Exhaustive evaluation of all query-path combinations becomes computationally prohibitive for realistic deployments. To address this, *ECO-LLM Emulator* uses adaptive Stratified Budget Allocation (SBA), outlined in Algorithm 1. The approach operates in two stages: first, it selects representative queries from each query type and evaluates all paths on these queries to identify high-performing paths per type; second, for remaining queries, it evaluates only the most promising paths identified in the initial stage. The budget factor B provides unified control over this two-stage exploration, independently scaling query selection for breadth ($B \times \sqrt{|Q|}$ queries) and path selection for depth ($B \times \sqrt{|\mathcal{P}|}$ paths per query). This sublinear scaling reduces total evaluations from $O(|Q| \times |\mathcal{P}|)$ to $O(\sqrt{|Q|} \times |\mathcal{P}| + |Q| \times \sqrt{|\mathcal{P}|})$, enabling exploration within computational constraints.

Representative queries are selected using k-means clustering on sentence transformer embeddings, choosing queries closest to each cluster centroid to ensure semantic diversity within each query type. Path ranking uses accuracy as the primary criterion, with cost or latency as secondary tiebreaker depending on optimization strategy (λ in Section 3.3.2).

ECO-LLM Emulator implements prefix caching to reduce redundant computation. When two paths differ only in their final component (e.g., both use step-back prompting and HyDE retrieval but different models), the system caches intermediate results and only executes the component that differs. This reduces computation by 30-50% in domains with expensive retrieval operations. For each evaluated path, *ECO-LLM Emulator* collects performance metrics (accuracy, latency, cost) for use by the *ECO-LLM Runtime*.

Algorithm 1 Adaptive Stratified Budget Allocation

Require: Queries Q, Paths $\mathcal{P}$, Budget factor B
Ensure: Evaluation results R
1: $Q_{rep} \leftarrow \text{StratifiedSample}(Q, B)$ *// k-means on embeddings by type*
2: Evaluate all paths $\mathcal{P}$ for queries Q_{rep}
3: Group and rank paths by accuracy, then cost/latency per query type
4: $k \leftarrow \max(1, B \cdot \sqrt{|\mathcal{P}|})$ *// Paths per remaining query*
5: **for** $q \in Q \setminus Q_{rep}$ **do**
6: $P_{select} \leftarrow k$ paths from top-ranked for q's type + random samples
7: Evaluate only paths in P_{select} for query q
8: **end for**

3.3 *ECO-LLM Runtime*

3.3.1 Problem Formulation

The *ECO-LLM Runtime* addresses path selection from Section 3.1 (Equation 6): for each unseen query, select a path maximizing accuracy while meeting latency and cost SLO constraints. The key challenge is that accuracy $\mathcal{A}(q, P)$ is unknown for new queries and the solution space grows exponentially with components. The *ECO-LLM Runtime* solves this through three stages: Critical Component Analysis (CCA) identifies which components critically affect accuracy for each query type, Domain-Specific Query Encoding (DSQE) learns to predict component requirements for unseen queries, and Runtime Path Selection (RPS) chooses optimal paths meeting SLO constraints in real-time.

3.3.2 Critical Component Analysis (CCA)

The CCA module identifies which subset of components are essential for individual queries through performance evaluation and ablation studies. The key insight is that for each query, only a subset of components critically impacts response quality.

Algorithm 2 outlines the CCA process. For each query (lines 2-12), we first identify the best-performing path P^* (line 3) using a lexicographic optimization approach that prioritizes accuracy first, then optimizes for either latency or cost among paths (controlled by $\lambda \in \{0, 1\}$ with comparable accuracy

Algorithm 2 Critical Component Analysis (CCA)

Require: Query set Q, Component options $O = \{M_q, M_r, M_c, M_m\}$, Accuracy threshold τ, Latency preference λ
Ensure: Critical component mapping $\Phi : Q \to 2^O$

1: $\Phi \leftarrow \emptyset$
2: **for** each $q \in Q$ **do**
3: $P^* \leftarrow \text{FindBestPath}(q, O, \lambda)$ *// Find path with highest accuracy*
4: $criticalComps \leftarrow \emptyset$
5: **for** each component type $t \in \{q, r, c, m\}$ **do**
6: $v \leftarrow P^*[t]$ *// Component value in best path*
7: $impactScore \leftarrow \text{CalculateImpact}(q, t, v, O, P^*)$
8: **if** $impactScore \geq \tau$ **then**
9: $criticalComps \leftarrow criticalComps \cup \{(t, v)\}$
10: **end if**
11: **end for**
12: $\Phi[q] \leftarrow criticalComps$
13: **end for**
14: **return** Φ

(within 1% tolerance). When $\lambda = 1$, the system optimizes for latency; when $\lambda = 0$, it optimizes for cost.

For each component type (lines 5-10), we calculate its impact on response quality:

$$\text{Impact}(q,t,v) = \mathcal{A}_{with}(q,t,v) - \mathcal{A}_{without}(q,t,v) \tag{7}$$

where $\mathcal{A}_{with}$ and $\mathcal{A}_{without}$ represent the mean accuracy with component value v fixed versus using alternatives:

$$\mathcal{A}_{with}(q,t,v) = \frac{1}{|\mathcal{P}_{t=v}|} \sum_{P \in \mathcal{P}_{t=v}} \mathcal{A}(q,P) \tag{8}$$

$$\mathcal{A}_{without}(q,t,v) = \frac{1}{|\mathcal{P}_{t\neq v}|} \sum_{P \in \mathcal{P}_{t\neq v}} \mathcal{A}(q,P) \tag{9}$$

Subset of components with impact scores exceeding threshold τ are marked as critical (lines 8-9). The result is a mapping Φ from queries to their critical components (line 11), which is used to train our domain-specific query encoder.

3.3.3 Domain-Specific Query Encoding (DSQE)

CCA identifies K distinct critical component sets across training queries. Each set specifies which implementations and configurations from the path tuple (Section 3.1) critically affect query accuracy—for instance, one set might be {(HyDE, θ_r), (LLMLingua, θ_c), (gpt-4o, θ_m)} while another is {(BasicRAG, θ'_r), (null, θ_c), (llama-8b, θ'_m)}. DSQE learns to predict which of these K sets a new query requires. The challenge is that semantic similarity does not reliably indicate component requirements: queries like "Turn off the bedroom lights" and "Why won't bedroom lights turn off?" are semantically similar but require different components. DSQE addresses this by training prototype vectors representing each component set and learning a projection that maps queries to their appropriate prototypes based on domain-specific patterns rather than semantic similarity alone. DSQE comprises three components:

1. Base Embedding Generator: We use SentenceTransformer [37] to generate dense vector representations $e_q \in \mathbb{R}^d$ for each query q, capturing semantic content as a start.

2. Projection Network: We train a multilayer perceptron $f_\theta : \mathbb{R}^d \to \mathbb{R}^d$ that transforms semantic embeddings into a specialized space where queries requiring similar component sets cluster together:

$$f_\theta(e_q) = f_n(\ldots f_2(f_1(e_q))) \tag{10}$$

where each layer f_i applies:

$$f_i(x) = \text{ReLU}(\text{Dropout}(W_i x + b_i)) \tag{11}$$

with learnable parameters $\theta = \{W_i, b_i\}_{i=1}^n$. This projection learns domain-specific patterns that map queries to component requirements.

3. Component-Set Prototypes: We learn K prototype vectors $\{v_k\}_{k=1}^K$ where each $v_k \in \mathbb{R}^d$ represents one of the distinct critical component sets identified by CCA. These prototypes are learnable parameters optimized jointly with the projection network.

We optimize the projection network and prototype vectors using three complementary objectives:

$$\mathcal{L}_{total} = \mathcal{L}_{contrast} + \alpha \cdot \mathcal{L}_{diversity} + \beta \cdot \mathcal{L}_{reg} \tag{12}$$

The contrastive loss encourages queries to map close to their corresponding component set's prototype. The diversity loss promotes separation between different prototypes, preventing collapse into similar representations. The regularization term prevents overfitting. During training, each query is labeled with its critical component set from CCA, and the projection network learns to encode queries such that their similarity to prototypes indicates component requirements. For inference, the system projects the query embedding, computes similarities to all K prototypes, and selects the component set corresponding to the most similar prototype.

3.3.4 Runtime Path Selection (Online)

The RPS module leverages the trained DSQE to make real-time path selection decisions for incoming queries, balancing response quality with SLO constraints, as outlined in Algorithm 3.

Algorithm 3 comprises three key steps. First, the system projects the new query embedding using the trained DSQE model and identifies the most similar prototype (lines 1–3), immediately revealing critical component requirements based

Algorithm 3 Runtime Path Selection (RPS)

Require: New query q_{new}, Component options O, Trained DSQE f_θ, Prototype vectors $\{v_k\}$, Training data D, SLO constraints $(\mathcal{L}_{max}, \mathcal{C}_{max})$

Ensure: Selected path $P_{selected}$

1: $e_{new} \leftarrow f_\theta(e_{q_{new}})$ *// Project query embedding*
2: $k^* \leftarrow \arg\max_k \text{sim}(e_{new}, v_k)$ *// Find nearest prototype*
3: $criticalComps \leftarrow \text{ComponentSet}(k^*)$
4: $validPaths \leftarrow \emptyset$
5: **for** each $P \in \mathcal{P}$ **do**
6: **if** $\mathcal{L}(P) \leq \mathcal{L}_{max}$ **and** $\mathcal{C}(P) \leq \mathcal{C}_{max}$ **and** $criticalComps \subseteq P$ **then**
7: $validPaths \leftarrow validPaths \cup \{P\}$
8: **end if**
9: **end for**
10: **if** $validPaths = \emptyset$ **then**
11: **return** $\text{DefaultPath}(criticalComps, O, D, \mathcal{L}_{max}, \mathcal{C}_{max})$
12: **end if**
13: **for** each $P \in validPaths$ **do**
14: $pathScores[P] \leftarrow \text{ScorePath}(P, e_{new}, D)$
15: **end for**
16: **return** $\arg\max_{P \in validPaths} pathScores[P]$

on patterns from training queries. Second, path filtering (lines 4–9) generates valid paths that meet both SLO constraints and critical component requirements:

$$\begin{aligned} \mathcal{P}_{valid} = \{P \in \mathcal{P} \mid & \mathcal{L}(P) \leq \mathcal{L}_{max} \wedge \\ & \mathcal{C}(P) \leq \mathcal{C}_{max} \wedge \\ & criticalComps \subseteq P\} \end{aligned} \tag{13}$$

Third, path scoring (lines 12–14) uses nearest neighbor analysis on training queries:

$$\text{Score}(P, q_{new}) = \sum_{q \in NN_k(q_{new})} w_q \cdot \mathbb{I}[P_q = P] \cdot \mathcal{A}(q, P_q) \tag{14}$$

where $NN_k(q_{new})$ returns k nearest training queries, w_q is the similarity weight, $\mathbb{I}$ is the indicator function, and P_q is the best path for query q. The system selects the path with the highest score (line 15).

For out-of-distribution queries with no valid paths (lines 10–11), a fallback mechanism selects components based on global performance statistics while respecting critical components, choosing options that exceed an accuracy threshold τ_{acc} while minimizing cost.

4 Implementation

We implemented both *ECO-LLM Emulator* and *ECO-LLM Runtime* as frameworks designed to simplify integration into development workflows. Our implementation focuses on maintainability, extensibility, and practical deployment. We plan to open-source the code upon publication.

We developed *ECO-LLM Emulator*, a 5k-line command-line application in Python, for experiment execution, context generation, and path evaluation. We implemented a hierarchical configuration system that separates module-specific configurations from global experiment parameters. Each emulation configuration receives a unique build identifier to track associated parameters, results, and artifacts. This design supports multiple parallel configurations for different scenarios or environments, and asynchronous execution enables parallel evaluation across compute resources.

We implemented key components across all module managers. For query processing, we use stepback prompting [51] and LLMLingua [19] for model-based query compression. Our context retrieval includes basic RAG [25] and HyDE [21]. For context processing, we use model-based compression [19] and Corrective-RAG [42] to improve accuracy. We integrate Ollama [32] for edge deployment and OpenAI [4] for cloud processing. We extended DeepEval [2] for LLM evaluation (similar to G-Eval metrics [27]), employing an ensemble of two LLM judges (GPT-4o and Gemini-2.5-Flash) to compare generated responses against ground truth answers from our datasets. To mitigate self-preference bias [40], neither judge model appears as a cloud option in our inference pipeline.

We implemented *ECO-LLM Runtime* as an OpenAI compatible server (3k lines of code), specifically designed for edge deployment. Our implementation extends the standard OpenAI API specification by adding support for build identifiers, SLO specification parameters, system state reporting capabilities, and enhanced error logging for deployment debugging. Our implementation provides developers with the tools and documentation to integrate new components and target specific edge deployment requirements, while maintaining flexibility to handle diverse deployment scenarios and usage patterns.

5 Evaluation

Our evaluation aims to answer four research questions: (1) How does ECO-LLM perform across diverse edge hardware platforms? (2) Can ECO-LLM generalize across diverse application domains? (3) Which components of ECO-LLM are important for its overall performance? (4) Can ECO-LLM maintain SLOs under cost and latency constraints?

5.1 Experiment Setup

Datasets. We evaluate ECO-LLM on five diverse domains. For **automotive assistants**, we use 3,000 pages of technical content from seven manufacturers covering diagnostics, maintenance, and troubleshooting. For **smarthome assistant**, we use the MPMQA [47] dataset, comprising 250 pages of documents for 36 products. For **agriculture**, we use the AgriQA dataset [7] with 250 queries on crop management and equipment operation. For **technical support**, we sample 250 queries from TechQA [35] containing real-world sup-

port questions. For **IoT security**, we use 250 queries on threat detection and best practices from a smart device security dataset [18]. For automotive and smarthome assistant, we generate queries using our Context Generator; for other domains, we use existing queries and responses in the datasets directly. We measure response quality using G-Eval [27] with an ensemble of two LLM judges (GPT-4o and Gemini-2.5-Flash) to mitigate self-preference bias [40]—neither judge appears as a cloud model in our inference pipeline. We measure latency as Time to First Token (TTFT) and cost using OpenAI pricing ($/1k queries).

Hardware platforms. We evaluate ECO-LLM on four edge-hardware platforms spanning resource-constrained to high-performance options : Jetson Orin Nano (8GB RAM, 33 TOPS, 15 Watts, $200), M1 Pro (16GB RAM, 11 TOPS, 45 Watts, $1,000), M4 (32GB RAM, 38 TOPS, 65 Watts, $700), and RTX A4500 (20GB VRAM, 186 TFLOPS, 200 Watts, $1,300).

Baselines. We compare against RouteLLM [33], a state-of-the-art model routing system, augmented with our RAG module for fair comparison. We evaluate three RouteLLM modes: RouteLLM-25, -50, and -75, with the number indicating percentage of queries sent to the cloud. We include GPT-4.1 as the cloud-only baseline and Oracle (exhaustive path selection) as the upper bound.

ECO-LLM configurations. We evaluate two practical configurations: ECO-LLM (Cost-First) minimizes operational cost, while ECO-LLM (Latency-First) optimizes latency. Both use three edge models (SmolLM2-1.7B [3], Llama3.2-3B [11], Phi-4 [36]) and three cloud options (GPT-4.1-nano, -mini, -full [4]), combined with query processing (step-back prompting [51]), retrieval (basic RAG [25], HyDE [21]), and context processing (Corrective-RAG [42], reranking [5]), yielding 200-300 distinct paths per domain.

5.2 Performance Across Edge Hardware

We evaluate cross-platform performance on the automotive and smart home domains, since they have different pre-processing requirements—automotive queries require extensive retrieval while smart home queries benefit minimally from pre-processing. Our results shown in Table 3 identifies M4-class platforms as a particularly cable edge device for deplopying responsive edge AI assistants given it integrated NPUs, 30-40 TOPS compute, and 20-32GB unified RAM within a 65W power envelope. As shown in Table 3, the M4 achieves responsive latencies with ECO-LLM (Latency-First) delivering 0.7s for automotive and 2.3s for smart home, while ECO-LLM (Cost-First) operates at $2.24-2.41 per 1k queries with 74-82% accuracy.

In comparison, the Jetson Orin Nano (8GB, 33 TOPS) has latencies exceeding 12s for automotive queries, making it unsuitable for interactive applications despite achieving higher accuracy through aggressive cloud routing. The M1 Pro (16GB, 11 TOPS), despite costing 43% more than the M4, provides lower compute (11 vs 38 TOPS) and has no meaningful advantage—ECO-LLM configurations achieve comparable accuracy and latency while the platform's higher cost reduces deployment flexibility.

Conversely, additional resources beyond M4-class yield diminishing returns. The RTX A4500 (186 TFLOPS, 200W) shows no latency advantage despite substantially higher power consumption—ECO-LLM (Latency-First) achieves 0.7s for automotive on both platforms. Furthermore, the A4500's power and cooling requirements limit deployment to infrastructure rich scenarios, whereas M4-class platforms enable deployment in vehicles, homes, and mobile contexts.

Across all platforms, ECO-LLM maintains consistent accuracy (73-85%) and minimal selection overhead (33-50ms), confirming that component selection strategies transfer across hardware. We focus our subsequent evaluation on the M4.

5.3 Generalization Across Domains

We evaluate ECO-LLM's ability to generalize across five diverse domains on the M4 hardware. Table 4 shows that ECO-LLM achieves lowest latency across all five domains and highest accuracy in three, using the same system configuration.

ECO-LLM (Latency-First) reduces response times substantially across domains: 1.3s for TechQA versus RouteLLM-75's 21s, 2.3s for Smart Home versus 22s, and 0.7s for Automotive versus 2.2s. ECO-LLM (Cost-First) achieves highest accuracy in Smart Home (74% vs RouteLLM-75's 66%), TechQA (84% vs 80%), and IoT Security (87% vs 85%). In Automotive and Agriculture, ECO-LLM configurations reach 77-82% accuracy compared to RouteLLM-75's 83-84%, while reducing costs by 4-18×—a tradeoff that may suit latency or cost-sensitive deployments.

In comparison, RouteLLM shows higher variance in accuracy across domains, ranging from 54% (Smart Home, R-25) to 85% (IoT, R-50/R-75). In Smart Home, RouteLLM-75 achieves only 66% accuracy with 22s latency; in TechQA, all RouteLLM configurations exhibit 20+ second latencies regardless of cloud routing percentage. These patterns suggest that model selection alone is insufficient when queries benefit from different preprocessing and retrieval strategies, which ECO-LLM's joint optimization addresses.

ECO-LLM provides predictable configuration options across all domains: Cost-First delivers cost-efficiency while Latency-First provides responsiveness, while keeping accuracy within 73-87% compared to RouteLLM's 54-85%. Practitioners can select based on deployment priorities without domain-specific tuning. The emulator discovers domain-appropriate component patterns automatically—the same training procedure yields different critical component configurations for each domain. Furthermore, the path selection overhead remains minimal (32-64ms).

Table 3: Performance across hardware platforms on automotive and smart home domains. M4-class platforms emerge as the practical deployment target—achieving sub-second latencies (0.7s automotive, 2.3s smart home) with 74-82% accuracy at $2.2-2.4 per 1k queries. Lower-tier platforms (Orin) exhibit prohibitive latencies (12+ seconds), while higher-tier platforms (A4500) show no latency advantage despite high rated power consumption. ECO-LLM maintains consistent accuracy (70-85%) and minimal selection overhead (28-50ms) across all platforms.

Hardware	Oracle	GPT-4.1	R-25	R-50	R-75	ECO-C	ECO-L
Automotive Assistant							
A4500	94 / 1.7 / 2.9	86 / 10.9 / 10.4	70 / 2.5 / 7.1	75 / 5.4 / 7.3	80 / 8.1 / 7.4	78 / 0.7 / 0.8(41)	82 / 3.1 / 0.7(37)
M4	95 / 1.7 / 4.1	89 / 12.3 / 1.0	73 / 3.5 / 4.3	80 / 7.3 / 3.0	84 / 9.9 / 2.2	82 / 2.4 / 1.2(50)	82 / 5.3 / 0.7(33)
M1Pro	94 / 1.6 / 3.9	88 / 11.0 / 15.2	72 / 2.2 / 14.8	78 / 5.5 / 14.1	82 / 8.3 / 13.4	79 / 1.6 / 1.1(48)	83 / 5.6 / 1.6(28)
Orin	96 / 1.8 / 14.7	98 / 12.9 / 41.7	66 / 0.3 / 54.1	68 / 0.7 / 54.1	70 / 1.9 / 53.2	85 / 5.2 / 16.6(45)	85 / 4.2 / 12.6(37)
Smart Home Assistant							
A4500	90 / 1.4 / 2.5	75 / 4.8 / 0.8	58 / 1.1 / 0.8	63 / 1.9 / 0.8	71 / 3.3 / 0.9	74 / 2.5 / 2.2(39)	75 / 2.6 / 2.0(37)
M4	91 / 1.9 / 4.6	73 / 8.8 / 24.8	54 / 2.0 / 22.6	59 / 3.4 / 22.6	66 / 5.9 / 22.0	74 / 2.2 / 4.4(36)	73 / 3.3 / 2.3(37)
M1Pro	90 / 1.7 / 3.2	75 / 4.8 / 0.8	56 / 1.6 / 6.8	61 / 2.7 / 6.8	68 / 4.6 / 6.8	74 / 0.7 / 3.1(36)	73 / 1.5 / 1.5(35)
Orin	90 / 1.8 / 9.3	75 / 4.8 / 0.9	58 / 1.1 / 1.1	64 / 1.9 / 1.2	71 / 3.3 / 1.0	70 / 3.0 / 3.9(34)	73 / 3.8 / 6.1(31)

Format: Accuracy (%) / Cost ($/1k queries) / Latency (seconds). Parentheses: ECO-LLM selection overhead (ms).
R-25/50/75: RouteLLM with 25%/50%/75% cloud routing (latency includes routing overhead). ECO-C/L: ECO-LLM (Cost/Latency-First).
Green: Best accuracy; Blue: Best cost; Mauve: Best latency among RouteLLM and ECO-LLM variants.

Table 4: Performance across five domains on M4 hardware. All baselines and GPT-4.1 use the best-average preprocessing configuration from emulation for fair comparison. ECO-LLM achieves lowest latency in all domains and highest accuracy in three, with consistent 73-87% accuracy compared to RouteLLM's 54-85% variance. Averaging across domains: ECO-LLM reduces costs by 61% ($2.74 vs $7.08) and latency by 6x (1.7s vs 10.6s) compared to RouteLLM-75 while maintaining comparable accuracy (79-81% vs 80%). Oracle outperforms GPT-4.1 (94% vs 85% average accuracy), demonstrating that joint optimization yields superior results even compared to the best cloud model with optimized preprocessing.

Domain	Oracle	GPT-4.1	R-25	R-50	R-75	ECO-C	ECO-L
Agriculture	96 / 0.6 / 3.1	87 / 5.8 / 1.0	80 / 1.1 / 1.6	82 / 2.3 / 1.5	83 / 3.6 / 1.3	79 / 0.2 / 1.4(32)	77 / 0.3 / 1.2(43)
TechQA	95 / 6.5 / 11.5	87 / 15.5 / 18.0	66 / 4.6 / 21.9	74 / 8.6 / 21.5	80 / 11.8 / 21.0	84 / 4.1 / 5.3(64)	81 / 3.7 / 1.3(48)
IoT Security	94 / 1.2 / 3.4	90 / 7.1 / 6.3	82 / 1.8 / 6.6	85 / 3.3 / 6.6	85 / 4.2 / 6.6	87 / 4.8 / 5.7(44)	84 / 4.4 / 3.1(40)
Automotive	95 / 1.7 / 4.1	89 / 12.3 / 1.0	73 / 3.5 / 4.3	80 / 7.3 / 3.0	84 / 9.9 / 2.2	82 / 2.4 / 1.2(50)	82 / 5.3 / 0.7(33)
Smart Home	91 / 1.9 / 4.6	73 / 8.8 / 24.8	54 / 2.0 / 22.6	59 / 3.4 / 22.6	66 / 5.9 / 22.0	74 / 2.2 / 4.4(36)	73 / 3.3 / 2.3(37)

Format: Accuracy (%) / Cost ($/1k queries) / Latency (seconds). Parentheses: ECO-LLM selection overhead (ms).
R-25/50/75: RouteLLM with 25%/50%/75% cloud routing (latency includes routing overhead). ECO-C/L: ECO-LLM (Cost/Latency-First).
Green: Best accuracy; Blue: Best cost; Mauve: Best latency among RouteLLM and ECO-LLM variants.

5.4 Contributions of ECO-LLM Components

We conduct systematic ablation studies to evaluate the necessity of ECO-LLM's adaptive runtime selection and the role of its components—Critical Component Analysis (CCA) and Domain-Specific Query Encoding (DSQE). We compare three configurations across all five domains on M4 hardware:

Config 1 - Static Policy (No CCA, No DSQE): Represents the traditional deployment approach where practitioners select a single "best average" query resolution path based on training data performance, then apply it uniformly to all queries. The path is chosen by (1) filtering paths within an accuracy margin of the best-performing configuration, then (2) selecting the lowest-cost or lowest-latency option based on deployment priority. This baseline requires no runtime selection logic.

Config 2 - CCA Only (No DSQE): Introduces runtime per-query path selection using full CCA to identify critical components, but employs direct 1-nearest-neighbor matching with sentence transformer embeddings instead of DSQE's learned projection. This configuration isolates DSQE's contribution while demonstrating basic adaptive selection (selection overhead: 20-30ms).

Config 3 - Full ECO-LLM (CCA + DSQE): The full ECO-LLM system with both CCA and DSQE's learned query encoding, as evaluated earlier (selection overhead: 30-50ms).

As shown in Table 5, Static (Config 1) Cost-First averages $1.5 per 1k queries across all the datasets but requires 10.2s latency—unsuitable for interactive applications. Conversely, Static (Config 1) Latency-First achieves 1.3s latency but costs $6.1, reflecting that a single fixed path cannot simultaneously serve queries with different component requirements. Introducing adaptive per-query selection addresses this limitation, but the encoding mechanism proves critical. The CCA-only configuration (Config 2)—which identifies

Table 5: Ablation study validating ECO-LLM components. Static policies sacrifice secondary metrics (Cost-First: 10.2s latency; Latency-First: $6.1 cost). CCA alone degrades accuracy; DSQE's learned encoding is necessary to optimize both dimensions simultaneously.

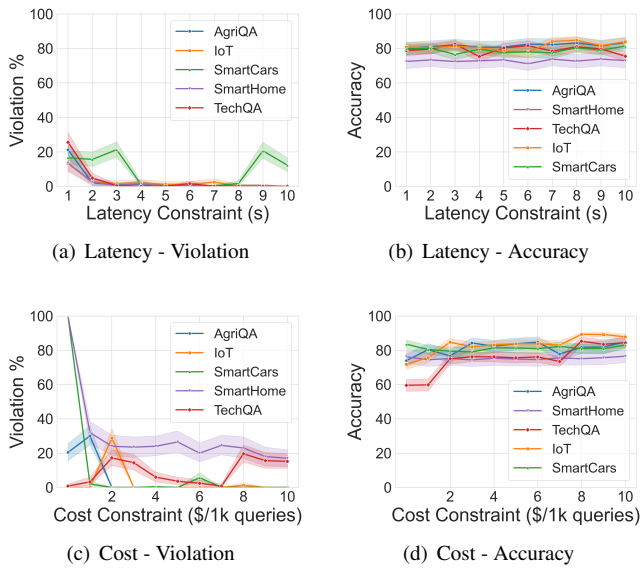
Domain	Cost-First Optimization			Latency-First Optimization		
	Static	CCA Only	**ECO-C**	Static	CCA Only	**ECO-L**
AgriQA	81.3 / 0.3 / 7.3	76.0 / 0.5 / 5.1	**79.0 / 0.2 / 1.4**	84.5 / 0.3 / 0.7	76.2 / 0.7 / 4.3	**77.0 / 0.3 / 1.2**
IoT Security	88.8 / 2.2 / 8.1	85.4 / 4.0 / 5.3	**87.0 / 4.8 / 5.7**	85.2 / 5.7 / 2.9	83.5 / 4.8 / 4.4	**84.0 / 4.4 / 3.1**
SmartCars	82.7 / 0.7 / 17.2	78.6 / 1.7 / 4.7	**82.0 / 2.4 / 1.2**	88.6 / 12.3 / 1.0	78.2 / 4.1 / 2.6	**82.0 / 5.3 / 0.7**
SmartHome	70.3 / 0.3 / 0.7	72.3 / 1.1 / 3.6	**74.0 / 2.2 / 4.4**	75.1 / 4.8 / 0.9	74.1 / 2.5 / 2.8	**73.0 / 3.3 / 2.3**
TechQA	85.2 / 4.1 / 17.8	82.5 / 7.6 / 14.9	**84.0 / 4.1 / 5.3**	86.8 / 7.4 / 0.9	82.5 / 10.5 / 9.7	**81.0 / 3.7 / 1.3**
Average	81.7 / 1.5 / 10.2	79.0 / 3.0 / 6.7	**81.2 / 2.7 / 3.6**	84.0 / 6.1 / 1.3	78.9 / 4.5 / 4.8	**79.4 / 3.4 / 1.7**

Format: Accuracy (%) / Cost ($/1k queries) / Latency (seconds).
Static: Fixed path for all queries; CCA Only: CCA with semantic matching (no DSQE).

critical components correctly but relies on 1-nearest-neighbor matching with semantic embeddings—actually degrades accuracy compared to static policies: 79.0% versus 81.7% for Cost-First, 78.9% versus 84.0% for Latency-First. This occurs because semantic similarity does not reliably indicate component requirements; queries with similar surface forms may need different pre-processing and retrieval strategies, as noted in Section 3.3. DSQE's learned projection resolves this mismatch. Full ECO-LLM restores accuracy to 81.2% and 79.4% respectively while substantially improving secondary metrics—Cost-First latency drops from 10.2s to 3.6s (3×), Latency-First cost drops from $6.1 to $3.4 (1.8×). By learning domain-specific patterns that map queries to appropriate component sets, DSQE enables effective per-query adaptation that static policies or naive semantic matching can't achieve.

5.5 SLO Attainment

We evaluate ECO-LLM's ability to meet user-defined Service Level Objectives (SLO) across all five domains on the M4 hardware. Figure 4 shows violation rates and accuracy as constraints vary from strict to relaxed—cost from $0-10 per 1k queries, latency from 1-10 seconds. ECO-LLM achieves near-zero violations when constraints are achievable. For latency SLOs (Figures 4(a),4(b)), AgriQA, IoT, SmartHome, and TechQA drop below 5% violation rate at 2-second constraints and approach zero beyond 4 seconds. For cost SLOs (Figures4(c), 4(d)), AgriQA, IoT, and SmartCars reach near-zero violations at $3-4 per 1k queries. Some domain-constraint combinations show persistent violations—SmartHome shows 15-25% cost violations even at $10, and TechQA shows 15-20% violations at higher cost constraints. These reflect ECO-LLM's design choice to prioritize response quality over strict SLO compliance. As described in Section 3.3, the runtime first identifies critical components for accuracy, then selects the lowest-cost or lowest-latency path with those components. When critical components inherently exceed the constraint, ECO-LLM exceeds the SLO rather than using cheaper alternatives that reduce accuracy. We believe that this design is



(a) Latency - Violation (b) Latency - Accuracy

(c) Cost - Violation (d) Cost - Accuracy

Figure 4: SLO attainment across four domains. Figures (a) and (b) show latency SLO results for ECO-LLM (Latency-First). Figures (c) and (d) show cost SLO results for ECO-LLM (Cost-First). Shaded regions represent 99% confidence intervals.

appropriate for domain-specific assistants where incorrect responses provide no practical value regardless of cost or speed. The accuracy plots (Figures 4(b), 4(d)) validate this approach: accuracy remains stable at 73-85% across all constraint levels. Strict SLO compliance would instead show accuracy degradation as constraints tighten. Furthermore, persistent violations are a signal to practitioners that their constraints are too restrictive for the domain's query distribution—information that silent quality degradation would obscure.

5.6 Exploration Budget Efficiency

We evaluate whether SBA's stratified sampling provides sufficient exploration when computational resources constrain exhaustive evaluation. Table 6 shows accuracy differences

Table 6: Accuracy delta from full (100%) exploration across varying exploration budgets. SBA maintains near-equivalent accuracy with reduced exploration—at 70% (B=10), accuracy stays within 1.5 points; at 35% (B=5), most domains remain within 3 points. Green indicates constrained exploration outperformed full exploration.

Domain	Cost-First			Latency-First		
	14%	35%	70%	14%	35%	70%
AgriQA	−0.7	−1.2	+3.3	−3.6	+0.3	−1.5
IoT Security	−8.1	−7.2	+0.1	+2.3	+0.5	+0.4
Automotive	−2.0	+0.6	+0.5	−7.2	−0.1	−0.6
Smart Home	—	−0.1	+0.8	−4.5	−4.5	−0.04
TechQA	−2.5	−2.4	−0.1	+2.9	+3.5	+1.2

14% (B=2), 35% (B=5), 70% (B=10): percentage of queries explored.
Values: accuracy delta from 100% exploration (percentage points).

relative to full exploration across three reduced budget levels—14% (B=2), 35% (B=5), and 70% (B=10) of queries explored.

At 70% exploration, accuracy remains within 1.5 percentage points of full exploration across all domains and optimization strategies, confirming that reduced exploration introduces minimal degradation. At 35% exploration, most domain-strategy combinations stay within 3 percentage points—AgriQA, Automotive, and TechQA show differences of 2.4% or less for both strategies, while IoT Security (Cost-First) and Smart Home (Latency-First) exhibit larger gaps of 7.2% and 4.5% respectively. These outliers suggest domains with more heterogeneous query distributions benefit from additional exploration.

The results exhibit non-monotonic patterns—several configurations show positive deltas where constrained exploration outperforms full exploration. TechQA Latency-First achieves 3.5% higher accuracy at 35% exploration than at 100%, and IoT Security Latency-First shows consistent improvements across all reduced budgets. This suggests that exhaustive exploration can discover paths that marginally improve training performance but generalize poorly, whereas SBA's clustering-based selection focuses on representative queries and yields more robust strategies. Overall, SBA provides effective subsampling when exploration budget is constrained. Practitioners prioritizing reliability can use 70% exploration for near-equivalent performance, while those with tighter constraints can use 35% with the understanding that complex domains may see modest accuracy reductions.

6 Discussion

ECO-LLM follows a domain-specific deployment model where the system trains once per domain and deploys a runtime optimized for that domain's characteristics. We discuss several considerations for deployment and future directions.

Query Distribution and Concept Drift. DSQE handles unseen queries within the training distribution but performance degrades for queries that significantly deviate from training patterns. This stems from DSQE's reliance on learned prototypes representing component sets observed during training. When query distributions shift over time—such as new vehicle features or device types not in original documentation—the critical component patterns may change, requiring retraining with updated documentation and regenerated queries. The performance gap between seen and unseen queries suggests potential for improved generalization through better query feature engineering and more sophisticated metric prediction models.

Hardware Portability and Device-Specific Profiling. The *ECO-LLM Emulator* profiles component performance on a single hardware configuration, producing platform-specific latency and cost metrics. Deploying to different edge devices requires profiling on target hardware, using virtual environments with configurable resources, or employing latency prediction models like nn-Meter to translate metrics across devices. Incorporating hardware specifications as runtime parameters and developing cross-device performance transfer methods would enhance ECO-LLM's applicability across diverse platforms.

Continuous Adaptation and Online Learning. The current implementation uses one-shot training where both CCA and DSQE train offline and remain fixed during deployment. CCA identifies critical component sets through ablation studies on training queries, and DSQE learns to predict these sets for new queries. In production, actual query distributions may differ from generated training queries, and performance requirements may evolve. While the runtime adapts path selection to changing SLO constraints, the offline training phase (CCA and DSQE) cannot adjust to new query patterns without complete retraining. Developing online learning mechanisms that continuously update both modules based on real-world feedback would enable adaptation to evolving workloads. This requires tracking low-confidence predictions, identifying emerging patterns, re-evaluating component criticality for new query types, and incrementally updating prototype representations.

7 Conclusion

We present ECO-LLM, a framework for building efficient domain-specific edge AI assistants that optimizes the complex interplay between query resolution components. ECO-LLM enables systematic evaluation of resolution paths and intelligent runtime adaptation while requiring minimal domain expertise from users. Our evaluation demonstrates that ECO-LLM achieves comparable performance to existing solutions while operating at the Pareto-optimal front for both cost and latency, as well as meeting user-defined SLOs across different domains. This combination of systematic optimization and runtime adaptability establishes ECO-LLM as a comprehensive solution for deploying and managing edge AI assistants in real-world applications.

References

[1] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. Automix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963*, 2023.

[2] Confident AI. Deepeval: The llm evaluation framework. https://github.com/confident-ai/deepeval, 2024.

[3] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM2: When Smol goes big – data-centric training of a small language model, 2025.

[4] Valentina Alto. *Modern Generative AI with ChatGPT and OpenAI Models: Leverage the capabilities of OpenAI's LLM for productivity and innovation with GPT3 and GPT4*. Packt Publishing Ltd, 2023.

[5] Nicholas Ampazis. Improving rag quality for large language models with topic-enhanced reranking. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 74–87. Springer, 2024.

[6] Apple. Apple intelligence. https://www.apple.com/apple-intelligence/, Retrieved on 2024-11.

[7] MSU CECO. Agxqa: Agricultural question answering dataset. https://huggingface.co/datasets/msu-ceco/agxqa_v1, 2024.

[8] Shuhao Chen, Weisen Jiang, Baijiong Lin, James T. Kwok, and Yu Zhang. RouterDC: Query-Based Router by Dual Contrastive Learning for Assembling Large Language Models, September 2024. arXiv:2409.19886 [cs].

[9] Jasper Dekoninck, Maximilian Baader, and Martin Vechev. A unified approach to routing and cascading for llms. *arXiv preprint arXiv:2410.10347*, 2024.

[10] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. *arXiv preprint arXiv:2404.14618*, 2024.

[11] Aaron Grattafiori et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Gemma Team et al. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*, 2024.

[13] Marah Abdin et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[14] Jia Fu, Xiaoting Qin, Fangkai Yang, Lu Wang, Jue Zhang, Qingwei Lin, Yubo Chen, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. Autorag-hp: Automatic online hyper-parameter tuning for retrieval-augmented generation. *arXiv preprint arXiv:2406.19251*, 2024.

[15] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2024.

[16] Google. Gemini makes your mobile device a powerful AI assistant. https://blog.google/products/gemini/made-by-google-gemini-ai-updates/, Retrieved on 2024-11.

[17] Shimon Ifrah. Getting started with azure openai.

[18] IoTSmart. Smart home iot security dataset. https://github.com/IoTSmart-art/smarthome, 2024.

[19] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*, 2023.

[20] Hongpeng Jin and Yanzhao Wu. CE-CoLLM: Efficient and Adaptive Large Language Models Through Cloud-Edge Collaboration, June 2025. arXiv:2411.02829 [cs].

[21] Marten Jostmann and Hendrik Winkelmann. Evaluation of hypothetical document and query embeddings for information retrieval enhancements in the context of diverse user queries. 2024.

[22] Rishi Kalra, Zekun Wu, Ayesha Gulley, Airlie Hilliard, Xin Guan, Adriano Koshiyama, and Philip Treleaven. Hypa-rag: A hybrid parameter adaptive retrieval-augmented generation system for ai legal and policy applications. *arXiv preprint arXiv:2409.09046*, 2024.

[23] Stefanos Laskaridis, Stylianos I. Venieris, Alexandros Kouris, Rui Li, and Nicholas D. Lane. The future of consumer Edge-AI computing. *IEEE Pervasive Comput.*, 23(3):21–30, 2024.

[24] Hojae Lee, Junho Kim, and SangKeun Lee. Mentor-kd: Making small language models better multi-step reasoners. *arXiv preprint arXiv:2410.09037*, 2024.

[25] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[26] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[27] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023.

[28] Noah Martin, Abdullah Bin Faisal, Hiba Eltigani, Rukhshan Haroon, Swaminathan Lamelas, and Fahad Dogar. Llmproxy: Reducing cost to access large language models. *arXiv preprint arXiv:2410.11857*, 2024.

[29] Microsoft. Copilot+ PC. https://www.microsoft.com/en-us/windows/business/devices/copilot-plus-pcs, Retrieved on 2024-11.

[30] Alireza Mohammadshahi, Arshad Rafiq Shaikh, and Majid Yazdani. Routoo: Learning to Route to Large Language Models Effectively, October 2024. arXiv:2401.13979 [cs].

[31] Quang H Nguyen, Duy C Hoang, Juliette Decugis, Saurav Manchanda, Nitesh V Chawla, and Khoa D Doan. Metallm: A high-performant and cost-efficient dynamic framework for wrapping llms. *arXiv preprint arXiv:2407.10834*, 2024.

[32] Ollama. Ollama. https://github.com/ollama/ollama, 2024.

[33] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[34] Deepak Babu Piskala, Vijay Raajaa, Sachin Mishra, and Bruno Bozza. Dynamic LLM Routing and Selection based on User Preferences: Balancing Performance, Cost, and Ethics. *International Journal of Computer Applications*, 186(51):1–7, November 2024. arXiv:2502.16696 [cs].

[35] Ganesh Rajagopalan et al. Techqa: A dataset for technical question answering. https://huggingface.co/datasets/rojagtap/tech-qa, 2024.

[36] Microsoft Research. Phi-4: Technical report, 2024.

[37] Marco Siino. All-mpnet at semeval-2024 task 1: Application of mpnet for evaluating semantic textual relatedness. In *Proceedings of the 18th International Workshop on Semantic Evaluation (SemEval-2024)*, pages 379–384, 2024.

[38] Dimitris Stripelis, Zijian Hu, Jipeng Zhang, Zhaozhuo Xu, Alay Dilipbhai Shah, Han Jin, Yuhang Yao, Salman Avestimehr, and Chaoyang He. Tensoropera router: A multi-model router for efficient llm inference, 2024.

[39] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Teola: Towards end-to-end optimization of llm-based applications. *arXiv preprint arXiv:2407.00326*, 2024.

[40] Koki Wataoka, Tsubasa Takahashi, and Ryokan Ri. Self-preference bias in llm-as-a-judge, 2025.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[42] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. Corrective retrieval augmented generation. *arXiv preprint arXiv:2401.15884*, 2024.

[43] Bufang Yang, Lixing He, Neiwen Ling, Zhenyu Yan, Guoliang Xing, Xian Shuai, Xiaozhe Ren, and Xin Jiang. EdgeFM: Leveraging Foundation Model for Open-set Learning on the Edge. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, pages 111–124, Istanbul Turkiye, November 2023. ACM.

[44] Zheming Yang, Yuanhao Yang, Chang Zhao, Qi Guo, Wenkai He, and Wen Ji. PerLLM: Personalized Inference Scheduling with Edge-Cloud Collaboration for Diverse LLM Services, May 2024. arXiv:2405.14636 [cs].

[45] Yuhang Yao, Han Jin, Alay Dilipbhai Shah, Shanshan Han, Zijian Hu, Yide Ran, Dimitris Stripelis, Zhaozhuo Xu, Salman Avestimehr, and Chaoyang He. Scalellm: A resource-frugal llm serving framework by optimizing end-to-end efficiency. *arXiv preprint arXiv:2408.00008*, 2024.

[46] Zhongzhi Yu, Zheng Wang, Yuhan Li, Haoran You, Ruijie Gao, Xiaoya Zhou, Sreenidhi Reedy Bommu, Yang Katie Zhao, and Yingyan Celine Lin. EDGE-LLM: Enabling Efficient Large Language Model Adaptation on Edge Devices via Layerwise Unified Compression and Adaptive Layer Tuning and Voting, June 2024. arXiv:2406.15758 [cs].

[47] Liang Zhang, Anwen Hu, Jing Zhang, Shuo Hu, and Qin Jin. Mpmqa: multimodal question answering on product manuals. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 13958–13966, 2023.

[48] Mingjin Zhang, Jiannong Cao, Xiaoming Shen, and Zeyang Cui. Edgeshard: Efficient llm inference via collaborative edge computing. *arXiv preprint arXiv:2405.14371*, 2024.

[49] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2024.

[50] Zesen Zhao, Shuowei Jin, and Z Morley Mao. Eagle: Efficient training-free router for multi-llm inference. *arXiv preprint arXiv:2409.15518*, 2024.

[51] Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H Chi, Quoc V Le, and Denny Zhou. Take a step back: Evoking reasoning via abstraction in large language models. *arXiv preprint arXiv:2310.06117*, 2023.

[52] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *CoRR*, abs/2308.07633, 2023.

[53] Richard Zhuang, Tianhao Wu, Zhaojin Wen, Andrew Li, Jiantao Jiao, and Kannan Ramchandran. EmbedLLM: Learning Compact Representations of Large Language Models, October 2024. arXiv:2410.02223 [cs].